# Use cases classification:

**Simple:**

Delete Account

View Delivery History

Access Customer Support

Access Account

**Moderate:**

Track Delivery

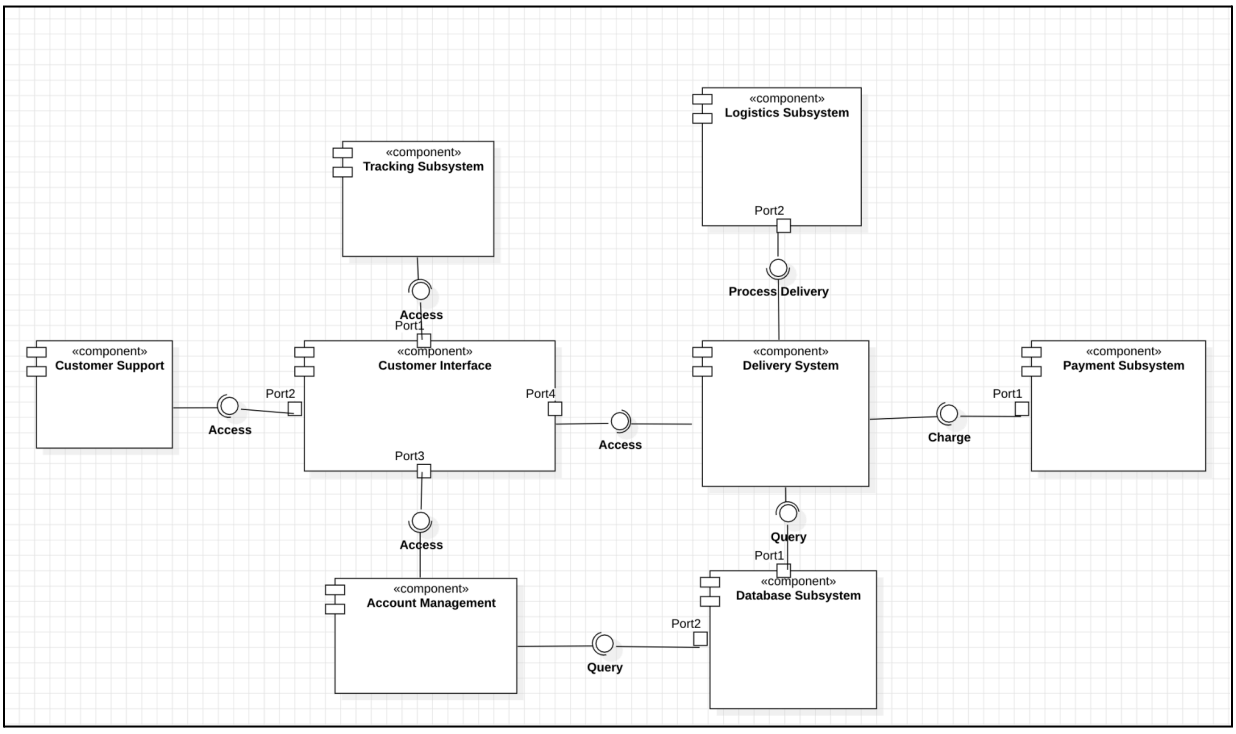
Cancel Delivery

Edit Account Information

**Complex:**

Delivery Request

# Architectural model




## Description

IComponent diagram –

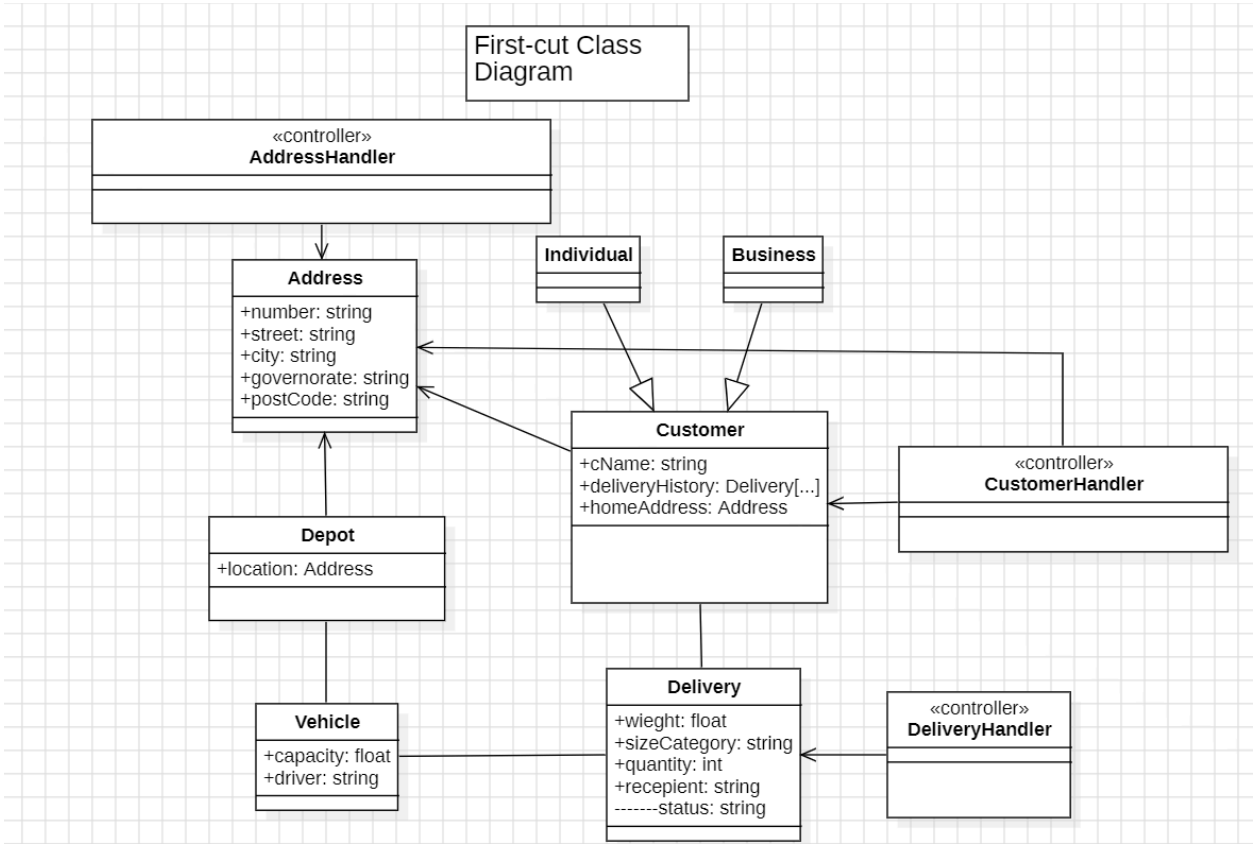
IA type of design diagram that shows the overall system architecture and the logical components within it for how the system will be implemented

IIdentifies the logical, reusable, and transportable system components that define the system architecture

IThe essential element of a component diagram is the component element with its API.

After completing the analysis and requirements models in the previous phase, we started the design phase with building an architectural model of the delivery management system for the business by way of a UML component diagram.
The purpose of constructing this diagram is to show the overall system architecture and the logical components within it, showcasing the interactions between said components by way of the different ports used.
The diagram consists of eight components in total, four main components and four subsystem components. The four main components are the Delivery System, Customer Interface, Customer Support system, and Account Management system. These components interact with each other and with the four subsystem components; Tracking, Logistics, Payments, and Database.

# First Cut Class Diagram

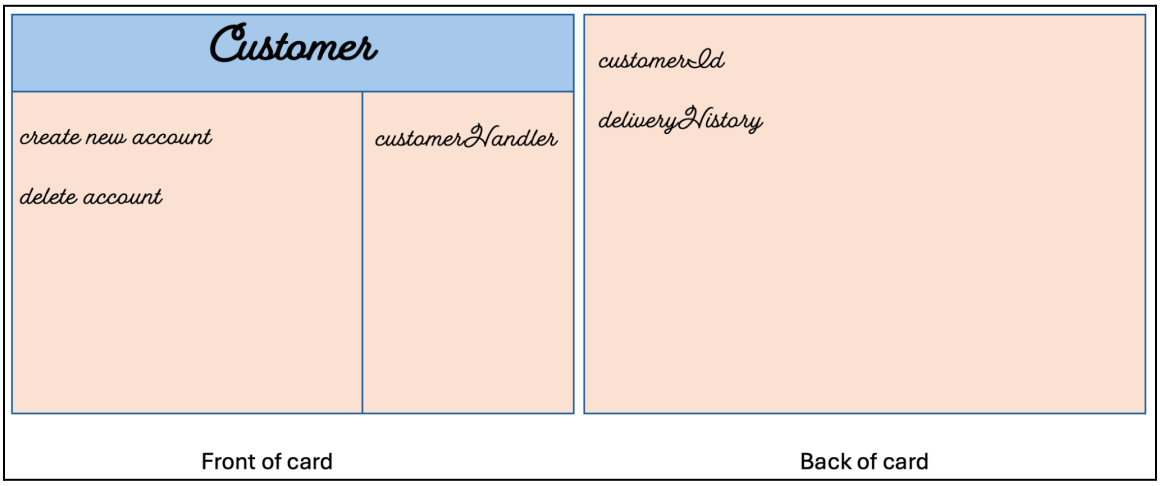

First-cut Class Diagram
«controller»
AddressHandler
Address
+number: string
+street: string
+city: string
+governorate: string
+postCode: string
Individual
Business
Customer
+cName: string
+deliveryHistory: Delivery[...]
+homeAddress: Address
«controller»
CustomerHandler
Depot
+location: Address
Vehicle
+capacity: float
+driver: string
Delivery
+wieght: float
+sizeCategory: string
+quantity: int
+recepient: string
------status: string
«controller»
DeliveryHandler

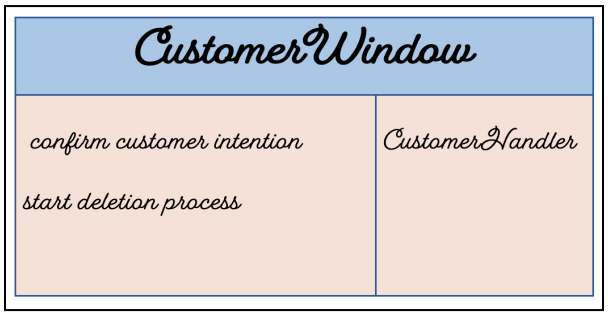
### Construction and Rationale

The first step in constructing the First-cut class diagram was to use the previously constructed domain model class diagram to start the design. The domain model class diagram was constructed in order to reflect the use cases in the application, ***use-case-by-use-case***. After the use-case driven domain model class diagram had been constructed, we picked the domain classes while checking the conditional requirements use-case-by-use-case. This led us to add two major additions to the classes.
The major addition was the “Address” class. This class is heavily used in a lot of use cases, like creating new user accounts, new delivery orders and pickup.
The last step in constructing the First-cut class diagram was to examine and elaborate each of the classes’ attributes, keeping in mind visibility requirements that would affect the relationships and associations between the classes. Here we note the addition of access modifiers in each class’ attribute; as well as the removal of multiplicities in the associations between the classes.

# CRC Cards

## Delete customer account



| CustomerHandler | | CustomerDB | |
|---|---|---|---|
| determine what data is to-be deleted (safe delete)<br><br>inform database | Customer | remove customer data | |

This use case revolves around the primary object object and how it interacts with the database that stores the customers' data, primarily during the interaction concerning when the customer wants to delete their account from our systems. The important attributes here are the customer's Id and their delivery history. These are to be used by the collaborating class, "CustomerHandler" to start the account deletion process.
Here, the user interacts with the window to confirm that the user does in fact want to delete their account, informing them of the risks and privileges they lose should that happen, and then initiate the deletion process.
After all of that, the "CustomerHandler" controller begins the deletion process by determining exactly what of the customer's data is to be deleted and what is to be kept and for how long, as some of the customer's data would remain relevant and important to the business after sometime of them deleting their account. The controller also examines the user's order history checking if there are any ongoing or pending deliveries; or outstanding payments that would prevent the account from being deleted.
Lastly, the controller informs the customer database of what exactly is to be done with the customer's data so the deletion process can be performed and finalized.

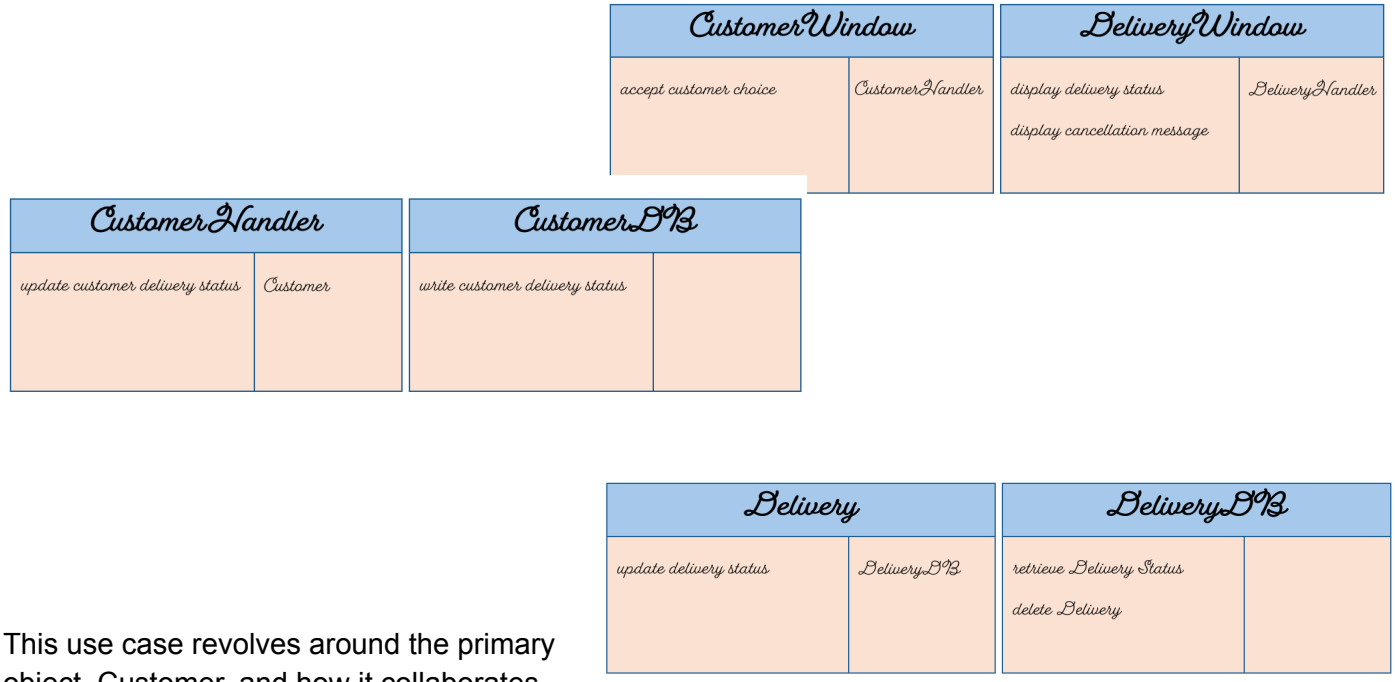
### Design Diagram Modifications

**Missing Responsibilities/Methods:**
CustomerDB: write(Customer), delete(Customer)
Adding: deleteCustomerAccount(customerID)

## View Delivery History

| Customer | | delivery Number<br>delivery Details |
|---|---|---|
| track delivery<br>cancel delivery | Delivery<br>CustomerDB | |

Front of card | Back of card

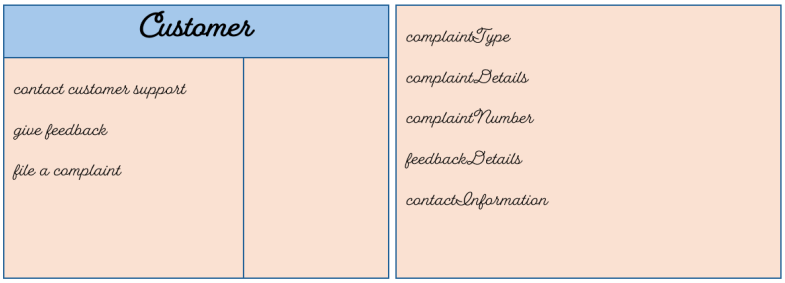



This use case revolves around the primary object, Customer, and how it collaborates with the Delivery class. Its responsibilities consist of tracking and canceling the customer's deliveries with the deliveryNumber and deliveryDetails attributes as the sent data between the two classes. There is a window for each of the customer and delivery classes, where the CustomerWindow accepts the customer choice to track or cancel a delivery, and the DeliveryWindow displays the delivery status in case of the tracking or displays a cancellation message if a delivery is canceled successfully. In addition, there is a handler for the customer class that updates the delivery status once a tracking or cancellation is requested. Furthermore, the data access classes like the CustomerDB, that writes the delivery status into the database, and the DeliveryDB, that retrieves the delivery status or deletes the delivery are responsible for managing the data sent between classes.

## Design Diagram Modifications

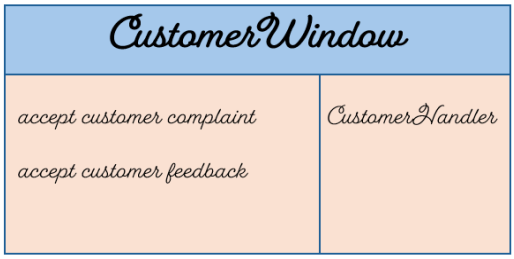
**Missing Responsibilities/Methods:**
Customer: trackDelivery(delivery) - cancelDelivery(delivery)
DeliveryWindow: displayStatus(delivery)
CustomerHandler: updateStatus(delivery)
CustomerDB: writeStatus(deliveryStatus) - DeliveryDB: retrieveStatus(deliveryStatus), deleteDelivery(delivery)

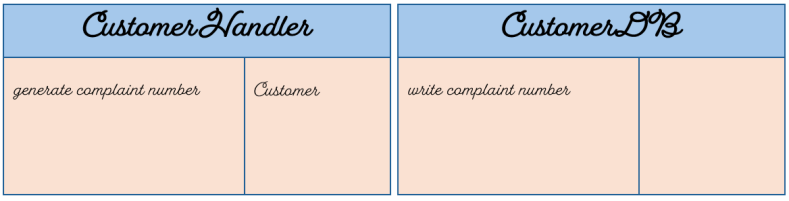
# Access Customer Support




Front of card

Back of card



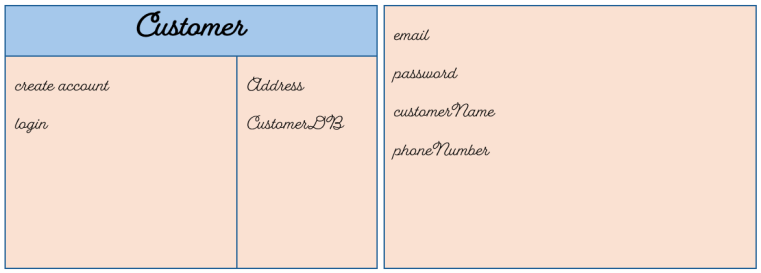




This use case has one primary class, which is the Customer class, that is responsible for contacting customer support, giving feedback, and filing complaints. Even though there are attributes that are used by this class, like data about the complaint, feedback, or the contact information of the customer support, it does not collaborate with any other classes, as it only sends messages that are not responded to immediately. There is only one window, the CustomerWindow, which accepts the customer's complaint or feedback and collaborates with the CustomerHandler. The CustomerHandler is responsible for generating a complaint number unique to the customer so that they can get back with the customer support through it, while the CustomerDB writes this complaint number into the database.

## Design Diagram Modifications

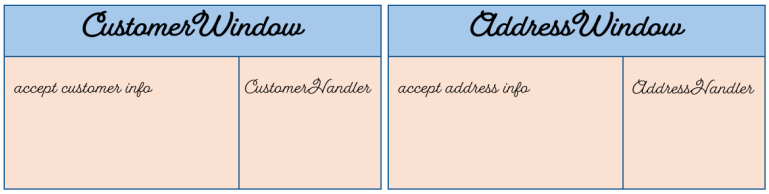
**Missing Responsibilities/Methods:**
Customer: giveFeedback(feedback), fileComplaint(complaint)
CustomerHandler: generateComplaintNumber(complaint)
CustomerDB: write(complaintNumber)

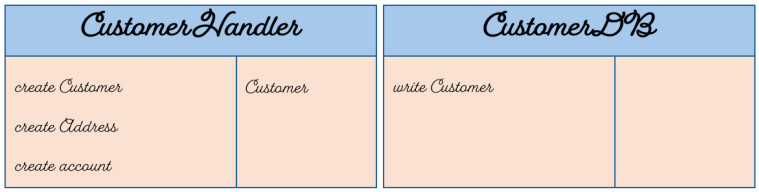
# Access Account




Front of card

Back of card



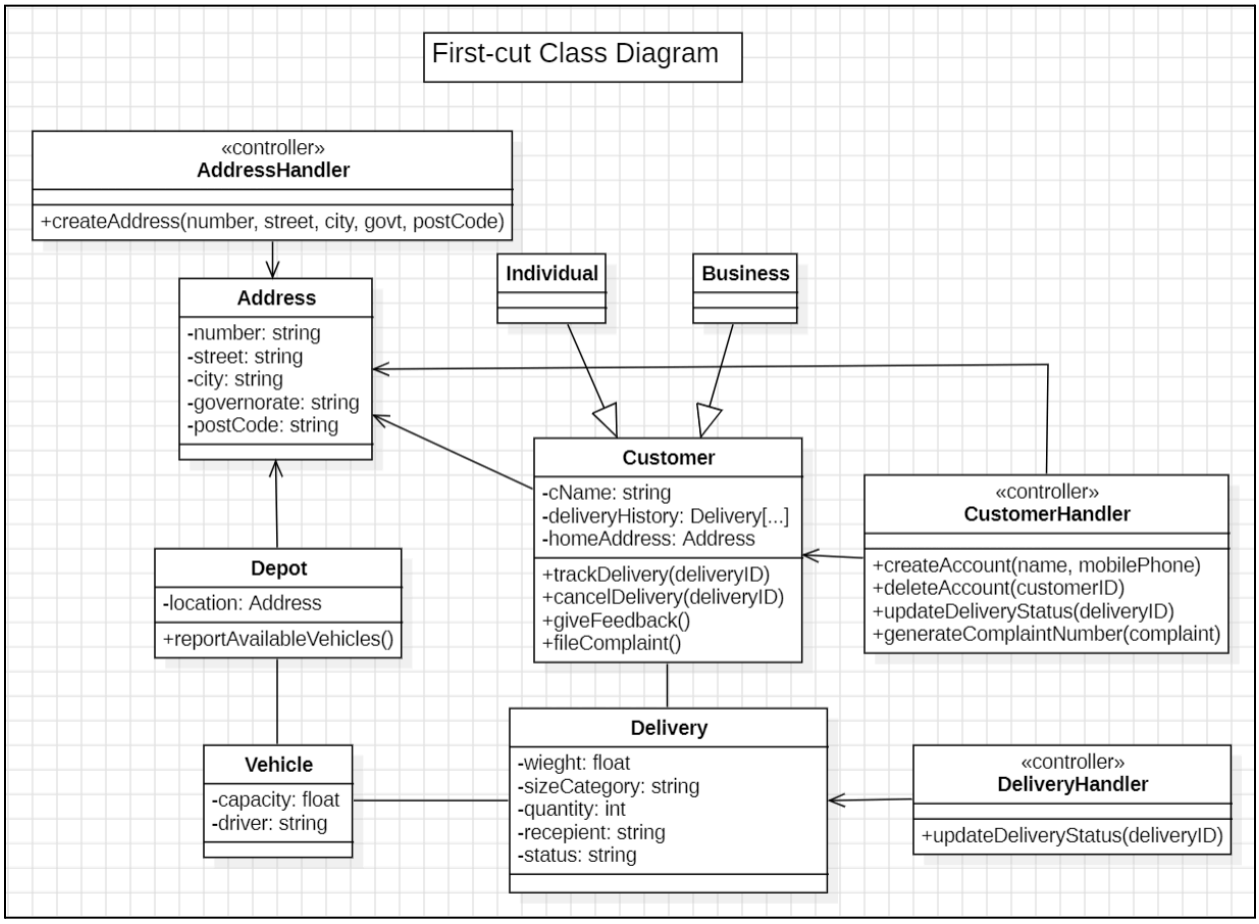


This use case involves two classes, the Customer class, which is the primary one, and its collaborator, the Address class. Once a customer creates an account, the Customer class does its responsibilities, which is to create an account or to login. The attributes used in these processes are email, password, customerName, and phoneNumber. There are two windows used in this use case, the CustomerWindow, which accepts the customer's information during registration or login, and the AddressWindow, which accepts the address information. The CustomerHandler creates the customer, the address, and the account from the provided data. In addition, the CustomerDB is responsible for writing the customer into the database, while the AddressDB writes the address.

## Design Diagram Modifications

**Missing Responsibilities/Methods:**
CustomerHandler: createAccount(customerInfo)
CustomerDB: write(Customer) - AddressDB: write(Address)

# First Cut Class Diagram (CRC-Refined)

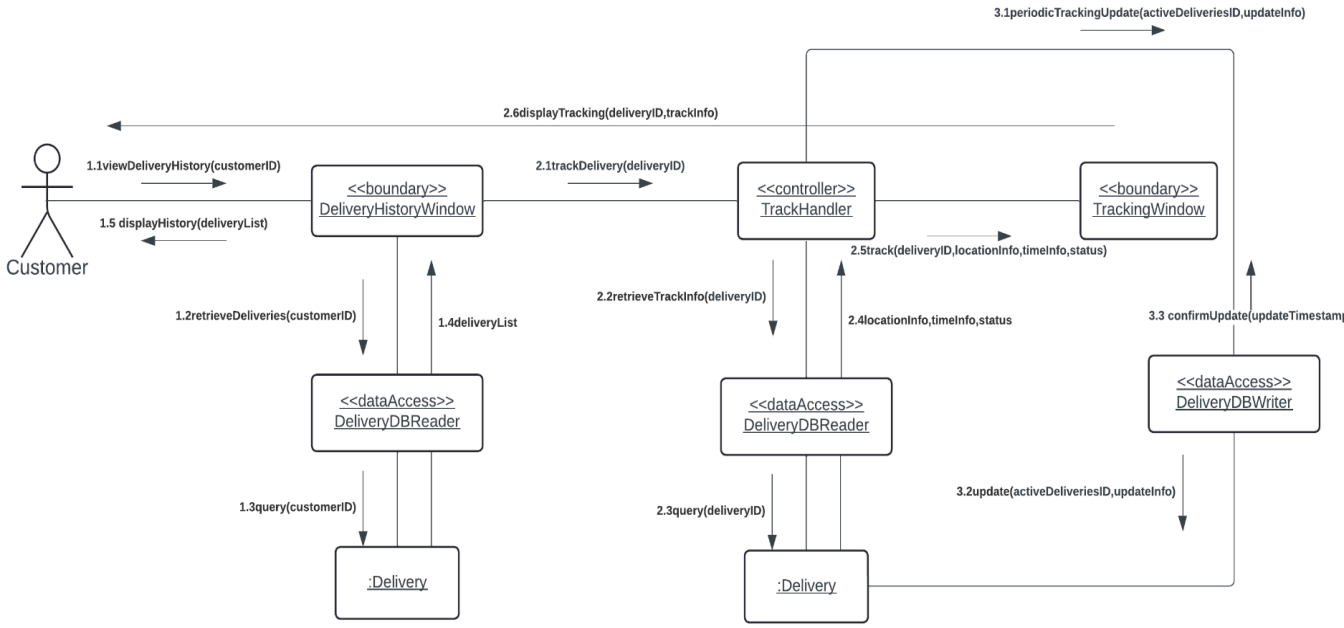



This FCCD is refined from the responsibilities (methods) that were present when we created the CRC, as requested per the deliverable document.
The major refinement was adding **controller classes** to be in charge of the use cases relating to each class. This is represented in the “CustomerHandler”, “DeliveryHandler”, and “addressHandler” controller classes, with their respective methods so far included.

Please note that the rest of the diagrams - as stated in the deliverable document - only are reported to show what needs to be updated to the FCCD to reach the final cut (design class diagram) after, but does not go to actually apply these changes (as is stated).

# Communication Diagrams

## Track Delivery




### Description

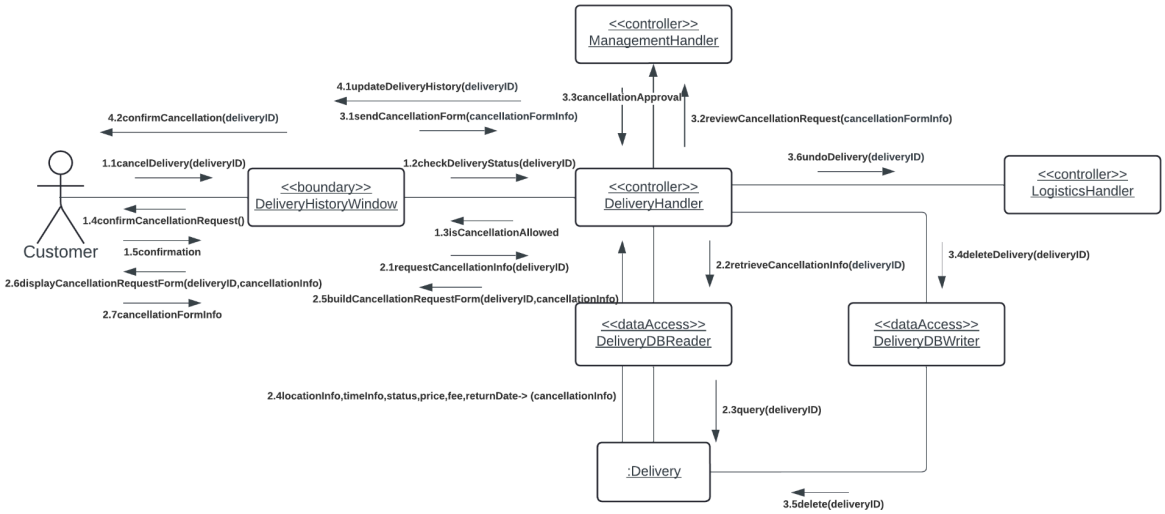
Customer accesses their delivery history list (including current deliveries), the boundary retrieves that list and displays it for the customer.
When the customer chooses to track an active delivery, track info for this delivery is retrieved from the database by the tracking subsystem, then displayed to the customer in the new tracking window.
Periodically, the tracking subsystem updates the tracking information for all the currently active deliveries to keep it up to date.

### Design Diagram Modifications

Missing classes with their attributes/methods:
**Data access:**
DeliveryDBReader: query().
DeliveryDBWriter: update(),confirmUpdate(updateTimestamp).
**Boundary:**
DeliveryHistoryWindow:deliveryList, retrieveDeliveries(customerID),displayHistory(deliveryList)).
TrackingWindow: displayTracking(deliveryID,trackInfo).

**Controller:**
TrackHandler:retrieveTrackInfo(deliveryID),track(trackInfo),
periodicTrackingUpdate(activeDeliveriesID, updateInfo).
viewDeliveryHistory should receive a customerID parameter.

# Cancel Delivery




## Description

The user -in the delivery history list page- requests a cancellation of an active delivery order, after checking that the order is able to be canceled. If so, delivery information is retrieved to build a cancellation form and is displayed to the customer. Then the customer fills the form, and it is sent to management for approval.
After being approved for cancellation, delivery order is deleted from the database and from the logistics system (drivers and vehicles are de-assigned etc..).
Lastly, the customer is displayed the confirmation message for the cancellation.

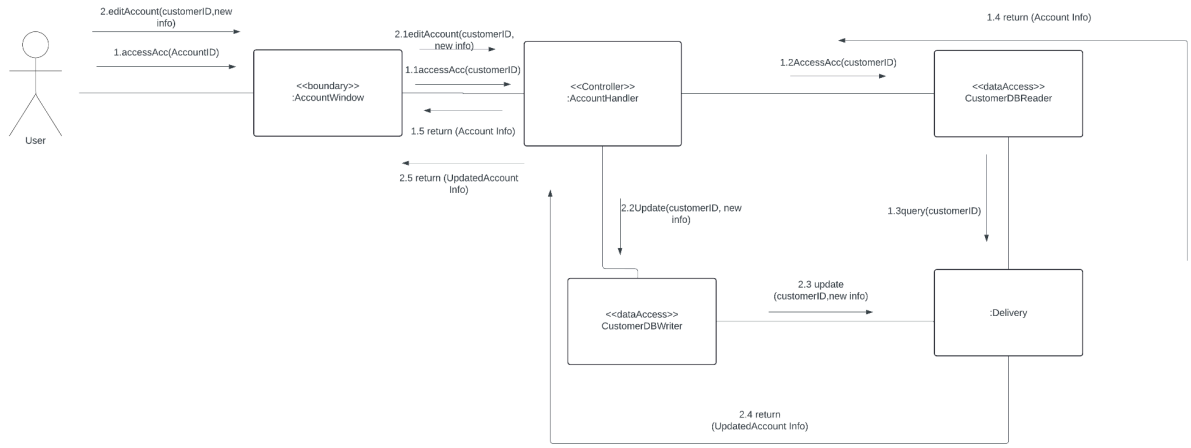
## Design Diagram Modifications

Missing classes with their attributes/methods:
**Data access:**
DeliveryDBReader: query().
DeliveryDBWriter: delete().
**Boundary:**
DeliveryHistoryWindow:checkDeliveryStatus(deliveryID), confirmCancellationRequest(isCancellationAllowed), requestCancellationInfo(deliveryID), displayCancellationRequestForm(deliveryID,cancellationInfo), cancellationFormInfo, sendCancellationForm(cancellationFormInfo), confirmCancellation(deliveryID).

**Controller:**
DeliveryHandler:
retrieveCancellationInfo(deliveryID),reviewCancellationRequest(cancellationFormInfo),
buildCancellationRequestForm(deliveryID,cancellationInfo),
undoDelivery(deliveryID),deleteDelivery(deliveryID).
ManagementHandler: approve(cancellationFormInfo).

LogisticsHandler is added, and should most likely separate DeliveryHandler into DeliveryHandler and LogisticsHandler. The first handles our deliveries from the top level (cancellation, info maintenance etc..) while the latter handles underlying processing of each delivery order (assignDelivery(), returnDelivery(), undoDelivery() etc..).

# Edit Account




## Description

The user clicks on Manage account and method accessAcc(...) is executed by forwarding the customerID to the AccountHandler and retrieving the account information from the database and returning it to be displayed on the AccountWindow , if the customer chooses to click on edit account they will enter their new information and the method editAccount(customerID,new info) is forwarded to the AccountHandler which then updates the record in the table Customer

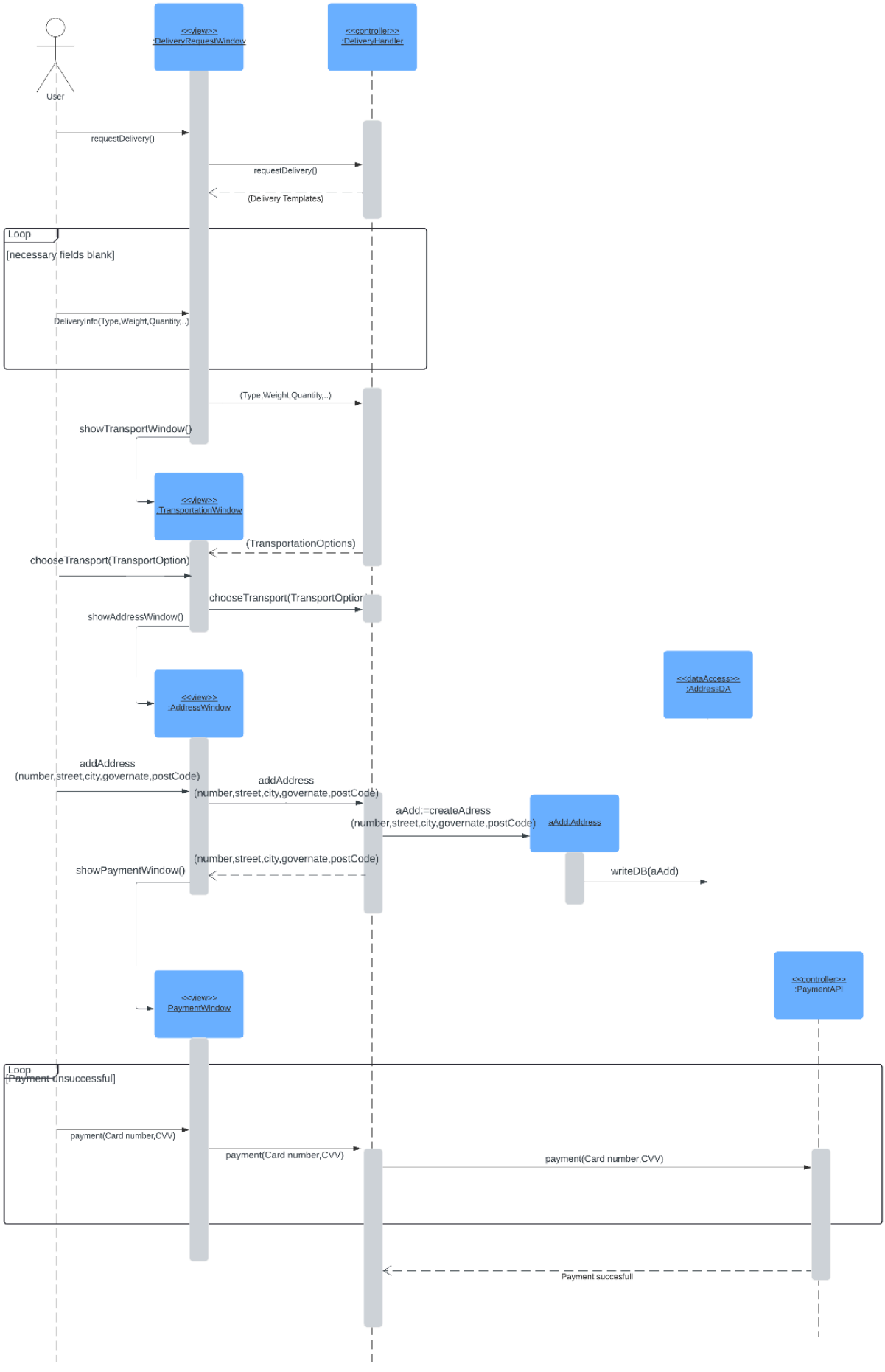
## Design Diagram Modifications

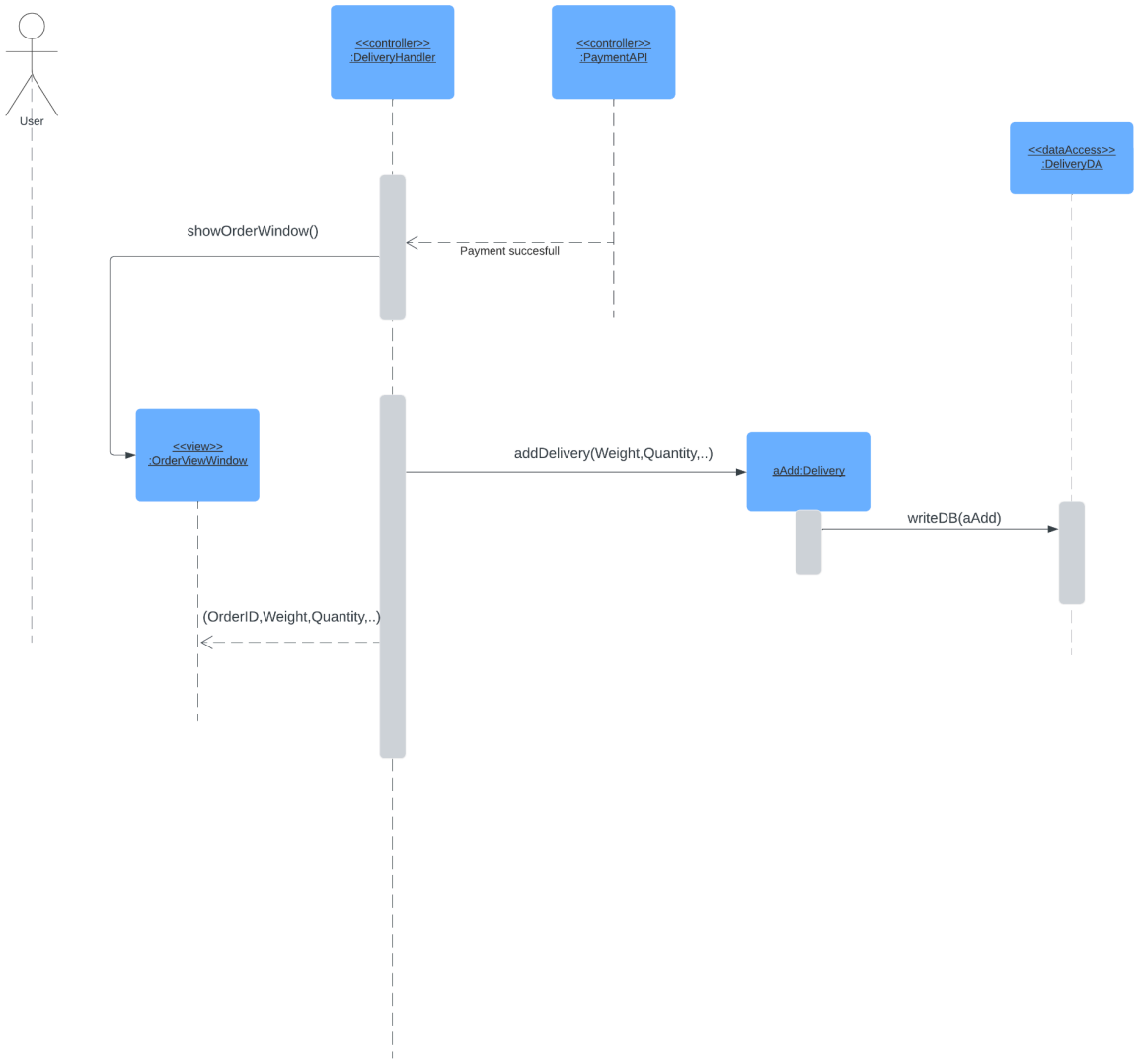
**Data access:**
AccountDBReader: query().
AccountDBWriter: update().

**Boundary:**
AccountWindow(): accessAcc(customerID) , editAccount(customerID,new info)
**Controller:**
AccountHandler: update(customerID,new info),accesAcc(customerID)

# Sequence Diagram

## Request Delivery


User
<<view>>
:DeliveryRequestWindow
<<controller>>
:DeliveryHandler
requestDelivery()
requestDelivery()
(Delivery Templates)
Loop
[necessary fields blank]
DeliveryInfo(Type,Weight,Quantity,..)
(Type,Weight,Quantity,..)
showTransportWindow()
<<view>>
:TransportationWindow
(TransportationOptions)
chooseTransport(TransportOption)
chooseTransport(TransportOption)
showAddressWindow()
<<dataAccess>>
:AddressDA
<<view>>
:AddressWindow
addAddress
(number,street,city,governate,postCode)
addAddress
(number,street,city,governate,postCode)
aAdd:=createAdress
(number,street,city,governate,postCode)
aAdd:Address
(number,street,city,governate,postCode)
showPaymentWindow()
writeDB(aAdd)
<<controller>>
:PaymentAPI
<<view>>
PaymentWindow
Loop
[Payment unsuccessful]
payment(Card number,CVV)
payment(Card number,CVV)
payment(Card number,CVV)
Payment succesfull




## Description

The customer clicks on the request delivery button on the homepage , they are then directed to the DeliveryRequestWindow where the DeliveryHandler would have retrieved the appropriate template the customer can choose from when filling the Delivery Information. The customer is forced to stay in the same page until all necessary fields have been filled and then the data is forwarded to the DeliveryHandler which returns the appropriate transportation options which the customer can choose from.The customer is then directed to the address where they enter their address which is then forwarded to the handler to be stored in table Address in the database.Finally the customer is directed to the payment page where the payment options will be displayed and once the customer’s transaction goes through confirmed by the payment API they will be directed to a page with their delivery summary and Order confirmation number

## Design Diagram Modifications

Missing classes with their attributes/methods:

**Data access:**

aAdd:Address :inserts record in table Address

aAdd:Delivery : inserts record in table Delivery

**View:**

DeliveryRequestWindow:requestDelivery(),showTransportWindow()

TransportWindow() :chooseTransport(TransportOption),showAddressWindow()

AddressWindow(): addAddress (number,street,city,governorate,postCode), showPaymentWindow()

PaymentWindow():payment(Card number,CVV) , showOrderWindow()

**Controller:**

DeliveryHandler: createAddress (number,street,city,governorate,postCode),payment(Card number,CVV)